Homework - 2

File Name and Exercises:

1. Exercise -1: exercise-1.ipynb
2. Exercise - 2: exercise-2.ipynb, exercise-2-a.ipynb (new sequence generator)
3. Exercise - 3: exercise-3.ipynb
4. Exercise - 4: exercise-4.ipynb
5. Exercise - 5: exercise-5.py,exercise-5b.py (Quickselect and graph plot)

Exercise 1:

Question: Task scheduler - a terrible randomised task scheduler creates and assigns tasks randomly to different queues. It decides on its own, whether to insert into each queue and stack a new item (unique ID) or to remove (delete). Currently, there are 10K time points, but you can also run it for 100K or 1M. It does not check much of anything. See code: HW2 Task Scheduler.ipynb

Simulate those operations on queues and stacks (collectively data structures, DS). Calculate the total performance of the scheduler and each DS separately. Report in a table, using one row for total nr and one row per DS with the following columns:

- How many items were inserted and deleted in total (and per DS)?
- How many erroneous commands (e.g. dequeue or pop from empty DS) does each DS receive?
- What maximal size does each DS reach and when?
- How many elements remain in each DS at the end of the scheduler work?

Answer: For Queue = 5; Stack = 3; MaxTime = 10000

Queue/ Stack No	Insert/Push	Delete/ Pop	Erroneous	Maximum Size	Maximum Time	Remaining Elements
Queue_1	3598	2686	7	912	9764	912
Queue_2	3704	2516	13	1188	9882	1188
Queue_3	3760	2464	6	1296	9887	1296
Queue_4	3936	2188	18	1748	9969	1748
Queue_5	4028	1894	12	2134	10000	2134
Stack_1	3685	2662	0	1023	9928	1023
Stack_2	3806	2310	3	1496	9933	1496
Stack_3	4021	1992	1	2029	10000	2029

Exercise 2:

a) **Question**: If items are remaining at the end of the run, dequeue or pop all remaining values one by one, decreasing the size of each DS by one at every next time point.

- Calculate for each DS, how long elements stay in it on average (including the new “emptying period” after the scheduler work?
- Which value stayed in each DS the longest, and for how long? (e.g. if item 3245 was inserted to Q3 at t00200 and removed at t02345 then it stayed in Q3 for 2145 units. Hint: add insertion time to DS together with item id).

Answer: For Queue = 5 and Stack = 3

Queue Results:

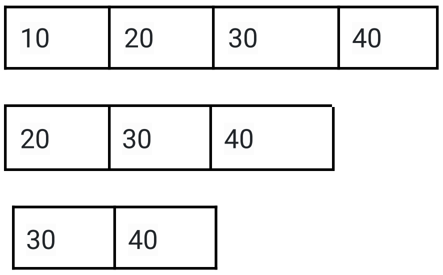
- Queue Q1: Avg time stayed = 2650.898276820456, Longest stay = 4551 (Item 16033)
- Queue Q2: Avg time stayed = 2648.188714902808, Longest stay = 4522 (Item 16159)
- Queue Q3: Avg time stayed = 2299.5175531914892, Longest stay = 3816 (Item 19014)
- Queue Q4: Avg time stayed = 2347.284044715447, Longest stay = 3527 (Item 20213)
- Queue Q5: Avg time stayed = 2079.707795431976, Longest stay = 3464 (Item 20477)

Stack Results:

- Stack S1: Avg time stayed = 2499.611668928087, Longest stay = 10970 (Item 37)
- Stack S2: Avg time stayed = 2389.5131371518655, Longest stay = 11257 (Item 226)
- Stack S3: Avg time stayed = 2107.8216861477244, Longest stay = 12022 (Item 3)

Question: Provide an interpretation of the main (expected or unexpected) similarities and differences between various data structures in DS.

Answer:
Queue: It implements ***FIFO*** operation. First in First Out



40

Stack: It implements ***LILO*** operation. Last in Last out.

10
20
30
40

10
20
30

10
20

10

Expected Similarities:

- Queue and Stack will have a similar average time for items under random insertion and deletions, especially when the scheduler operates for a long time
- Queue and Stack will show a similar error pattern like trying to dequeue or pop from an empty data structure

Expected Differences:

- The stack will have the longest staying period of an element since it follows the **LIFO** method
- The queue will have the shortest average period compared to the stack since it follows the **FIFO** method

Unexpected Differences:

- The larger the Queues, the number of erroneous commands decrease compared to the stack or vice versa, because the randomness probabilities are distributed

Results: ***For Queue= 5 and Stack= 3 and time =10000***

Queue/	**Insert/Push**	**Delete/**	**Erroneous**	**Maximum**	**Maximum**	**Remaining**

Stack No		Pop		Size	Time	Elements
Queue_1	3598	2686	7	912	9764	912
Queue_2	3704	2516	13	1188	9882	1188
Queue_3	3760	2464	6	1296	9887	1296
Queue_4	3936	2188	18	1748	9969	1748
Queue_5	4028	1894	12	2134	10000	2134
Stack_1	3685	2662	0	1023	9928	1023
Stack_2	3806	2310	3	1496	9933	1496
Stack_3	4021	1992	1	2029	10000	2029

```
Queue Results:
Queue Q1: Avg time stayed = 2728.733592880979, Longest stay = 4574 (Item 16245)
Queue Q2: Avg time stayed = 2409.596650192202, Longest stay = 4170 (Item 17820)
Queue Q3: Avg time stayed = 2616.8674443266173, Longest stay = 4101 (Item 18117)
Queue Q4: Avg time stayed = 2452.5641485275287, Longest stay = 3683 (Item 19782)
Queue Q5: Avg time stayed = 2156.9679471494983, Longest stay = 3522 (Item 20423)

Stack Results:
Stack S1: Avg time stayed = 2734.346817691478, Longest stay = 10658 (Item 511)
Stack S2: Avg time stayed = 2484.470633693972, Longest stay = 11388 (Item 159)
Stack S3: Avg time stayed = 2091.6250624063905, Longest stay = 11946 (Item 109)
```

For Queue = 3 and Stack = 5

Queue/ Stack No	Insert/Push	Delete/ Pop	Erroneous	Maximum Size	Maximum Time	Remaining Elements
Queue_1	3626	2653	1	973	9953	973
Queue_2	3897	2342	2	1555	9845	1555
Queue_3	3937	2342	2	1975	10000	1975
Stack_1	3619	2774	7	845	9852	845
Stack_2	3706	2568	5	1138	9739	1138
Stack_3	3759	2430	11	1329	9904	1329
Stack_4	3897	2259	15	1638	9936	1638

Stack_5	4097	1973	4	2124	10000	2124

Queue Results:
Queue Q1: Avg time stayed = 2558.0929398786543, Longest stay = 4323 (Item 17050)
Queue Q2: Avg time stayed = 2509.269951244547, Longest stay = 3863 (Item 18972)
Queue Q3: Avg time stayed = 1859.8956057912117, Longest stay = 3200 (Item 21543)

Stack Results:
Stack S1: Avg time stayed = 2618.8168002210555, Longest stay = 10509 (Item 302)
Stack S2: Avg time stayed = 2496.899352401511, Longest stay = 11016 (Item 85)
Stack S3: Avg time stayed = 2344.1308858739026, Longest stay = 10978 (Item 321)
Stack S4: Avg time stayed = 2296.888119065948, Longest stay = 11069 (Item 570)
Stack S5: Avg time stayed = 2062.618013180376, Longest stay = 12069 (Item 45)

Enhancing Fairness and Avoiding Errors:

1. Balancing insertions and deletions:
    a. Uniform distribution to insertion and deletion will prevent any single DS from being overloaded or underutilized
    b. Round-robin allocation to insert or delete the element
2. To avoid erroneous deletions:
    a. Check the DS before performing the deletion operation
    b. Keep track of what item is present and what is not
    c.

Comparison Analysis between improved sequence-generated operations and old sequence-generated functions:

Old Sequence generator:

Queue/ Stack No	**Insert/Push**	**Delete/ Pop**	**Erroneous**	**Maximum Size**	**Maximum Time**	**Remaining Elements**
Queue_1	3598	2686	7	912	9764	912
Queue_2	3704	2516	13	1188	9882	1188
Queue_3	3760	2464	6	1296	9887	1296
Queue_4	3936	2188	18	1748	9969	1748
Queue_5	4028	1894	12	2134	10000	2134
Stack_1	3685	2662	0	1023	9928	1023
Stack_2	3806	2310	3	1496	9933	1496
Stack_3	4021	1992	1	2029	10000	2029

New Sequence Generator:

Queue/ Stack No	**Insert/Push**	**Delete/ Pop**	**Erroneous**	**Maximum Size**	**Maximum Time**	**Remaining Elements**
Queue_1	5044	4945	2	99	9996	99
Queue_2	5054	4959	10	95	10000	95
Queue_3	4980	4969	75	11	9998	11
Queue_4	5035	4920	18	115	9999	115
Queue_5	4984	4957	16	27	10000	27
Stack_1	4860	4837	153	23	10000	23
Stack_2	4999	4949	84	50	9997	50
Stack_3	4996	4970	78	26	10000	26

After using the new sequence generator, the remaining elements in the queue and the stack have been significantly reduced

Exercise 3:
Question- the provided code compares sorting methods. It includes a Quicksort() implementation. Your task is to run the code for 1M random values. Characterise, what is wrong with the provided code? Fix it to solve that proble/issue. HW2 Quicksort.ipynb
After you have fixed it, run it for 3-4 more data types: small precision integers, very large integers, floating point numbers, and differently generated data just to see if there are any data type specific differences in speed.
Answer:
Issue with the provided code:

- While doing the partition specifically while checking the larger element than the pivot element, the right side(high side) of the pivot element was swapped but the left side (low side) wasn't swapped, causing the function to go for an infinite loop.

The code has been fixed and attached to the zip folder

```
Result:
# Built-in 1
Time to sort 1000000 elements: 0.56 seconds
Time to sort 1000000 elements: 0.63 seconds
Time to sort 1000000 elements: 0.62 seconds
# Built-in 2
Time to sort 1000000 elements: 0.61 seconds
Time to sort 1000000 elements: 0.08 seconds
Time to sort 1000000 elements: 0.07 seconds
# Quicksort 1
```

```
Time to sort 1000000 elements: 6.43 seconds
Time to sort 1000000 elements: 5.10 seconds
Time to sort 1000000 elements: 6.45 seconds
# Quicksort 2
Time to sort 1000000 elements: 5.16 seconds
Time to sort 1000000 elements: 26.95 seconds
Time to sort 1000000 elements: 192.24 seconds
# Quicksort 3 : Small Integers
Time to sort 12 elements: 0.00 seconds
# Quicksort 4 : Large Integers
Time to sort 10 elements: 0.00 seconds
# Quicksort 5 : Floating Numbers
Time to sort 10 elements: 0.00 seconds
# Quicksort 6: Random Data
Time to sort 100 elements: 0.00 seconds
```

From the results observed for different types of input data, there is no significant time difference in the performance of the quick sort algorithm based on data types such as small precision integers, very large integers, or floating-point numbers. The execution time largely remains consistent across these data types

Exercise 4:
Question: Implement the Quicksort with two pivots. Describe how you might try to avoid bad splits to make your code "unbreakable". Compare your “best” take on Quicksort to the code that was presented to you in Task 3. How many x faster can you make your implementation (e.g. compare and vary the ways you split data)?

Answer: The code has been implemented and provided in the zip folder. Kindly refer that
Result:
#QuickSort 1
Time to sort 1000000 elements: 5.37 seconds
Time to sort 1000000 elements: 5.44 seconds
Time to sort 1000000 elements: 5.33 seconds
#QuickSort 2
Time to sort 1000000 elements: 4.80 seconds
Time to sort 1000000 elements: 19.69 seconds
Time to sort 1000000 elements: 108.59 seconds
#Dual Quicksort 1
Time to sort 1000000 elements: 1.69 seconds
Time to sort 1000000 elements: 1.05 seconds
Time to sort 1000000 elements: 1.04 seconds
#Dual Quicksort 2
Time to sort 1000000 elements: 1.04 seconds
Time to sort 1000000 elements: 0.28 seconds
Time to sort 1000000 elements: 0.28 seconds

Comparing QuickSort and Dual Quicksort:

Calculating the average execution speed for each sorting,
Quicksort1: (5.37+5.44+5.33) / 3 = 5.38s
Quicksort2:(4.80+19.69+108.59)/3 = 44.36
Dual Quicksort1: (1.69 +1.05+1.04)/3 = 1.26
Dual Quicksort2: (1.04+0.28+0.28)/3 = 0.53

To Identify the **x** time faster: Quicksort/Dual Quicksort

- q1/dq1 => 5.38s/1.26 = 4.26 => dq1 is 4.26 faster then q1
- q2/dq2 => 44.36/0.53 = 83.6 => dq2 is 83.6 faster then q2

Avoiding the bad split: A bad split can cause the execution time longer and for the larger dataset it will lead to maximum recursion depth exceeding in comparison error. In order to avoid that we need to sort the array before calling the recursion, i.e we need to implement the recursion first to the shortest array part and then the largest part.

Exercise 5:

Binary Search and Order Statistics - how many random lookups and rank order statistics queries can be performed within 1 minute from 10M elements of sorted and unsorted data?
There are two types of queries: a) Look up the key for its membership and rank k. b) Look up, which value is ranked k'th in data. Generate the keys and ranks randomly.
Run these two types of queries on unordered data (with linear time methods) and on already ordered data (binary search and lookup).

From how many random rank order queries from originally randomly ordered data it would be actually faster to first sort and then "look up" the key using binary search?

Since the linear time order statistics query changes underlying data, how does that affect the speed after many queries over the data? Describe what happens to underlying data after repeated application of linear rank order queries. Visualize that gradual change using much smaller random data, e.g. 1000 values only.

Answer:

Linear Search: Iterates through each element in an array and finds the required input. It usually applied for unordered list

Binary Search: It is applied for ordered array(sorted). It finds the median of the array and checks whether the given input is greater or lesser than the median. If the median is smaller it will move the left side and if it's larger it will check the right side.

Query type and Performance Analysis:

a)Look up the key for its membership and rank k (Query type A)

1. Unordered Data:
   Membership Check: Linear Search is needed to check if the data is present in the list. So the time

complexity: O(n)
Rank Determination: Linear Search is needed to find the index of the given key. Time complexit is O(n)

2. Ordered Data:
   Membership Check: Binary search can be used to determine if the key exists, Time complexity isO(logn)
   Rank Determination: If the key is found, its rank is directly available, Time complexity for this case will be O(1)

b) Lookup which value is Ranked k' th in data (Query type B)

1. Unordered Data: A selection Algorithm like Quickselec is used, and time complexity will be O(n)
2. Ordered Data: Directly we can access the k'th element and time complexity will be O(1)

Performance Evaluation:

- Unordered Data:
  - Membership and Rank Lookup Time:
    - Total Time per Query: O(n)
    - Number of Queries in 1 minute = 60sec/ T(query)
      Where T(query) is the time taken per query)
    - 60/O(10000000) = 0.000006
- Ordered Data:
  - Membership check Time: O(log n)
  - Rank Lookup Time: O(1)
  - Number of queries in 1 minute = 60sec/Tquery

Sorting Time: O(n log n)

O(log n) for membership and O(1) for rank lookup

Number of queries for unordered= 60 sec/T(unordered)

Number of Queries for ordered = 60 sec - T(sort)/T(ordered)

Sorting becomes beneficial when,

60 sec - T(sort)/T(ordered) > 60 sec/T(unordered)

Effect of Linear Time order Statistics Queries on Data:

When performing linear-time order statistics queries, the data undergoes a partial sorting, this makes the data set sorted but not fully. So for the ordered data future queries might benefit from the reduced randomness but then it won't match the fully sorted data